EXHIBIT 5

Introduction to Algorithms, Second Edition

Thomas H. Cormen
Charles E. Leiserson
Ronald L. Rivest
Clifford Stein
The MIT Press
Cambridge , Massachusetts London, England
McGraw-Hill Book Company
Boston Burr Ridge , IL Dubuque , IA Madison , WI New York San Francisco St. Louis
Montréal Toronto

This book is one of a series of texts written by faculty of the Electrical Engineering and Computer Science Department at the Massachusetts Institute of Technology. It was edited and produced by The MIT Press under a joint production-distribution agreement with the McGraw-Hill Book Company.

Ordering Information:

North America

Text orders should be addressed to the McGraw-Hill Book Company. All other orders should be addressed to The MIT Press.

Outside North America

All orders should be addressed to The MIT Press or its local distributor.

First edition 1990

This book was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Introduction to algorithms / Thomas H. Cormen ... [et al.].-2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-262-03293-7 (hc.: alk. paper, MIT Press).-ISBN 0-07-013151-1 (McGraw-Hill)

1. Computer programming. 2. Computer algorithms. I. Title: Algorithms. II. Cormen, Thomas H.

QA76.6 I5858 2001

005.1-dc21

2001031277

Preface

This book provides a comprehensive introduction to the modern study of computer algorithms. It presents many algorithms and covers them in considerable depth, yet makes their design and analysis accessible to all levels of readers. We have tried to keep explanations elementary without sacrificing depth of coverage or mathematical rigor.

Each chapter presents an algorithm, a design technique, an application area, or a related topic. Algorithms are described in English and in a "pseudocode" designed to be readable by anyone who has done a little programming. The book contains over 230 figures illustrating how the algorithms work. Since we emphasize *efficiency* as a design criterion, we include careful analyses of the running times of all our algorithms.

The text is intended primarily for use in undergraduate or graduate courses in algorithms or data structures. Because it discusses engineering issues in algorithm design, as well as mathematical aspects, it is equally well suited for self-study by technical professionals.

In this, the second edition, we have updated the entire book. The changes range from the addition of new chapters to the rewriting of individual sentences.

To the teacher

This book is designed to be both versatile and complete. You will find it useful for a variety of courses, from an undergraduate course in data structures up through a graduate course in algorithms. Because we have provided considerably more material than can fit in a typical one-term course, you should think of the book as a "buffet" or "smorgasbord" from which you can pick and choose the material that best supports the course you wish to teach.

You should find it easy to organize your course around just the chapters you need. We have made chapters relatively self-contained, so that you need not worry about an unexpected and unnecessary dependence of one chapter on another. Each chapter presents the easier material first and the more difficult material later, with section boundaries marking natural stopping points. In an undergraduate course, you might use only the earlier sections from a chapter; in a graduate course, you might cover the entire chapter.

We have included over 920 exercises and over 140 problems. Each section ends with exercises, and each chapter ends with problems. The exercises are generally short questions that test basic mastery of the material. Some are simple self-check thought exercises, whereas others are more substantial and are suitable as assigned homework. The problems are more elaborate case studies that often introduce new material; they typically consist of several questions that lead the student through the steps required to arrive at a solution.

We have starred ($\star$) the sections and exercises that are more suitable for graduate students than for undergraduates. A starred section is not necessarily more difficult than an unstarred

one, but it may require an understanding of more advanced mathematics. Likewise, starred exercises may require an advanced background or more than average creativity.

To the student

We hope that this textbook provides you with an enjoyable introduction to the field of algorithms. We have attempted to make every algorithm accessible and interesting. To help you when you encounter unfamiliar or difficult algorithms, we describe each one in a step-by-step manner. We also provide careful explanations of the mathematics needed to understand the analysis of the algorithms. If you already have some familiarity with a topic, you will find the chapters organized so that you can skim introductory sections and proceed quickly to the more advanced material.

This is a large book, and your class will probably cover only a portion of its material. We have tried, however, to make this a book that will be useful to you now as a course textbook and also later in your career as a mathematical desk reference or an engineering handbook.

What are the prerequisites for reading this book?

- You should have some programming experience. In particular, you should understand recursive procedures and simple data structures such as arrays and linked lists.
- You should have some facility with proofs by mathematical induction. A few portions of the book rely on some knowledge of elementary calculus. Beyond that, Parts I and VIII of this book teach you all the mathematical techniques you will need.

To the professional

The wide range of topics in this book makes it an excellent handbook on algorithms. Because each chapter is relatively self-contained, you can focus in on the topics that most interest you.

Most of the algorithms we discuss have great practical utility. We therefore address implementation concerns and other engineering issues. We often provide practical alternatives to the few algorithms that are primarily of theoretical interest.

If you wish to implement any of the algorithms, you will find the translation of our pseudocode into your favorite programming language a fairly straightforward task. The pseudocode is designed to present each algorithm clearly and succinctly. Consequently, we do not address error-handling and other software-engineering issues that require specific assumptions about your programming environment. We attempt to present each algorithm simply and directly without allowing the idiosyncrasies of a particular programming language to obscure its essence.

To our colleagues

We have supplied an extensive bibliography and pointers to the current literature. Each chapter ends with a set of "chapter notes" that give historical details and references. The chapter notes do not provide a complete reference to the whole field of algorithms, however. Though it may be hard to believe for a book of this size, many interesting algorithms could not be included due to lack of space.

Despite myriad requests from students for solutions to problems and exercises, we have chosen as a matter of policy not to supply references for problems and exercises, to remove the temptation for students to look up a solution rather than to find it themselves.

Changes for the second edition

What has changed between the first and second editions of this book? Depending on how you look at it, either not much or quite a bit.

A quick look at the table of contents shows that most of the first-edition chapters and sections appear in the second edition. We removed two chapters and a handful of sections, but we have added three new chapters and four new sections apart from these new chapters. If you were to judge the scope of the changes by the table of contents, you would likely conclude that the changes were modest.

The changes go far beyond what shows up in the table of contents, however. In no particular order, here is a summary of the most significant changes for the second edition:

- Cliff Stein was added as a coauthor.
- Errors have been corrected. How many errors? Let's just say several.
- There are three new chapters:
  - Chapter 1 discusses the role of algorithms in computing.
  - Chapter 5 covers probabilistic analysis and randomized algorithms. As in the first edition, these topics appear throughout the book.
  - Chapter 29 is devoted to linear programming.
- Within chapters that were carried over from the first edition, there are new sections on the following topics:
  - perfect hashing (Section 11.5),
  - two applications of dynamic programming (Sections 15.1 and 15.5), and
  - approximation algorithms that use randomization and linear programming (Section 35.4).
- To allow more algorithms to appear earlier in the book, three of the chapters on mathematical background have been moved from Part I to the Appendix, which is Part VIII.
- There are over 40 new problems and over 185 new exercises.
- We have made explicit the use of loop invariants for proving correctness. Our first loop invariant appears in Chapter 2, and we use them a couple of dozen times throughout the book.
- Many of the probabilistic analyses have been rewritten. In particular, we use in a dozen places the technique of "indicator random variables," which simplify probabilistic analyses, especially when random variables are dependent.
- We have expanded and updated the chapter notes and bibliography. The bibliography has grown by over 50%, and we have mentioned many new algorithmic results that have appeared subsequent to the printing of the first edition.

We have also made the following changes:

- The chapter on solving recurrences no longer contains the iteration method. Instead, in Section 4.2, we have "promoted" recursion trees to constitute a method in their own right. We have found that drawing out recursion trees is less error-prone than iterating

recurrences. We do point out, however, that recursion trees are best used as a way to generate guesses that are then verified via the substitution method.

- The partitioning method used for quicksort (Section 7.1) and the expected linear-time order-statistic algorithm (Section 9.2) is different. We now use the method developed by Lomuto, which, along with indicator random variables, allows for a somewhat simpler analysis. The method from the first edition, due to Hoare, appears as a problem in Chapter 7.
- We have modified the discussion of universal hashing in Section 11.3.3 so that it integrates into the presentation of perfect hashing.
- There is a much simpler analysis of the height of a randomly built binary search tree in Section 12.4.
- The discussions on the elements of dynamic programming (Section 15.3) and the elements of greedy algorithms (Section 16.2) are significantly expanded. The exploration of the activity-selection problem, which starts off the greedy-algorithms chapter, helps to clarify the relationship between dynamic programming and greedy algorithms.
- We have replaced the proof of the running time of the disjoint-set-union data structure in Section 21.4 with a proof that uses the potential method to derive a tight bound.
- The proof of correctness of the algorithm for strongly connected components in Section 22.5 is simpler, clearer, and more direct.
- Chapter 24, on single-source shortest paths, has been reorganized to move proofs of the essential properties to their own section. The new organization allows us to focus earlier on algorithms.
- Section 34.5 contains an expanded overview of NP-completeness as well as new NP-completeness proofs for the hamiltonian-cycle and subset-sum problems.

Finally, virtually every section has been edited to correct, simplify, and clarify explanations and proofs.

Web site

Another change from the first edition is that this book now has its own web site: http://mitpress.mit.edu/algorithms/. You can use the web site to report errors, obtain a list of known errors, or make suggestions; we would like to hear from you. We particularly welcome ideas for new exercises and problems, but please include solutions.

We regret that we cannot personally respond to all comments.

Acknowledgments for the first edition

Many friends and colleagues have contributed greatly to the quality of this book. We thank all of you for your help and constructive criticisms.

MIT's Laboratory for Computer Science has provided an ideal working environment. Our colleagues in the laboratory's Theory of Computation Group have been particularly supportive and tolerant of our incessant requests for critical appraisal of chapters. We specifically thank Baruch Awerbuch, Shafi Goldwasser, Leo Guibas, Tom Leighton, Albert Meyer, David Shmoys, and Éva Tardos. Thanks to William Ang, Sally Bemus, Ray Hirschfeld, and Mark Reinhold for keeping our machines (DEC Microvaxes, Apple Macintoshes, and Sun

Sparcstations) running and for recompiling TEXwhenever we exceeded a compile-time limit. Thinking Machines Corporation provided partial support for Charles Leiserson to work on this book during a leave of absence from MIT.

Many colleagues have used drafts of this text in courses at other schools. They have suggested numerous corrections and revisions. We particularly wish to thank Richard Beigel, Andrew Goldberg, Joan Lucas, Mark Overmars, Alan Sherman, and Diane Souvaine.

Many teaching assistants in our courses have made significant contributions to the development of this material. We especially thank Alan Baratz, Bonnie Berger, Aditi Dhagat, Burt Kaliski, Arthur Lent, Andrew Moulton, Marios Papaefthymiou, Cindy Phillips, Mark Reinhold, Phil Rogaway, Flavio Rose, Arie Rudich, Alan Sherman, Cliff Stein, Susmita Sur, Gregory Troxel, and Margaret Tuttle.

Additional valuable technical assistance was provided by many individuals. Denise Sergent spent many hours in the MIT libraries researching bibliographic references. Maria Sensale, the librarian of our reading room, was always cheerful and helpful. Access to Albert Meyer's personal library saved many hours of library time in preparing the chapter notes. Shlomo Kipnis, Bill Niehaus, and David Wilson proofread old exercises, developed new ones, and wrote notes on their solutions. Marios Papaefthymiou and Gregory Troxel contributed to the indexing. Over the years, our secretaries Inna Radzihovsky, Denise Sergent, Gayle Sherman, and especially Be Blackburn provided endless support in this project, for which we thank them.

Many errors in the early drafts were reported by students. We particularly thank Bobby Blumofe, Bonnie Eisenberg, Raymond Johnson, John Keen, Richard Lethin, Mark Lillibridge, John Pezaris, Steve Ponzio, and Margaret Tuttle for their careful readings.

Colleagues have also provided critical reviews of specific chapters, or information on specific algorithms, for which we are grateful. We especially thank Bill Aiello, Alok Aggarwal, Eric Bach, Vašek Chvátal, Richard Cole, Johan Hastad, Alex Ishii, David Johnson, Joe Kilian, Dina Kravets, Bruce Maggs, Jim Orlin, James Park, Thane Plambeck, Hershel Safer, Jeff Shallit, Cliff Stein, Gil Strang, Bob Tarjan, and Paul Wang. Several of our colleagues also graciously supplied us with problems; we particularly thank Andrew Goldberg, Danny Sleator, and Umesh Vazirani.

It has been a pleasure working with The MIT Press and McGraw-Hill in the development of this text. We especially thank Frank Satlow, Terry Ehling, Larry Cohen, and Lorrie Lejeune of The MIT Press and David Shapiro of McGraw-Hill for their encouragement, support, and patience. We are particularly grateful to Larry Cohen for his outstanding copyediting.

Acknowledgments for the second edition

When we asked Julie Sussman, P.P.A., to serve as a technical copyeditor for the second edition, we did not know what a good deal we were getting. In addition to copyediting the technical content, Julie enthusiastically edited our prose. It is humbling to think of how many errors Julie found in our earlier drafts, though considering how many errors she found in the first edition (after it was printed, unfortunately), it is not surprising. Moreover, Julie sacrificed her own schedule to accommodate ours-she even brought chapters with her on a trip to the Virgin Islands! Julie, we cannot thank you enough for the amazing job you did.

The work for the second edition was done while the authors were members of the Department of Computer Science at Dartmouth College and the Laboratory for Computer Science at MIT. Both were stimulating environments in which to work, and we thank our colleagues for their support.

Friends and colleagues all over the world have provided suggestions and opinions that guided our writing. Many thanks to Sanjeev Arora, Javed Aslam, Guy Blelloch, Avrim Blum, Scot Drysdale, Hany Farid, Hal Gabow, Andrew Goldberg, David Johnson, Yanlin Liu, Nicolas Schabanel, Alexander Schrijver, Sasha Shen, David Shmoys, Dan Spielman, Gerald Jay Sussman, Bob Tarjan, Mikkel Thorup, and Vijay Vazirani.

Many teachers and colleagues have taught us a great deal about algorithms. We particularly acknowledge our teachers Jon L. Bentley, Bob Floyd, Don Knuth, Harold Kuhn, H. T. Kung, Richard Lipton, Arnold Ross, Larry Snyder, Michael I. Shamos, David Shmoys, Ken Steiglitz, Tom Szymanski, Éva Tardos, Bob Tarjan, and Jeffrey Ullman.

We acknowledge the work of the many teaching assistants for the algorithms courses at MIT and Dartmouth, including Joseph Adler, Craig Barrack, Bobby Blumofe, Roberto De Prisco, Matteo Frigo, Igal Galperin, David Gupta, Raj D. Iyer, Nabil Kahale, Sarfraz Khurshid, Stavros Kolliopoulos, Alain Leblanc, Yuan Ma, Maria Minkoff, Dimitris Mitsouras, Alin Popescu, Harald Prokop, Sudipta Sengupta, Donna Slonim, Joshua A. Tauber, Sivan Toledo, Elisheva Werner-Reiss, Lea Wittie, Qiang Wu, and Michael Zhang.

Computer support was provided by William Ang, Scott Blomquist, and Greg Shomo at MIT and by Wayne Cripps, John Konkle, and Tim Tregubov at Dartmouth. Thanks also to Be Blackburn, Don Dailey, Leigh Deacon, Irene Sebeda, and Cheryl Patton Wu at MIT and to Phyllis Bellmore, Kelly Clark, Delia Mauceli, Sammie Travis, Deb Whiting, and Beth Young at Dartmouth for administrative support. Michael Fromberger, Brian Campbell, Amanda Eubanks, Sung Hoon Kim, and Neha Narula also provided timely support at Dartmouth.

Many people were kind enough to report errors in the first edition. We thank the following people, each of whom was the first to report an error from the first edition: Len Adleman, Selim Akl, Richard Anderson, Juan Andrade-Cetto, Gregory Bachelis, David Barrington, Paul Beame, Richard Beigel, Margrit Betke, Alex Blakemore, Bobby Blumofe, Alexander Brown, Xavier Cazin, Jack Chan, Richard Chang, Chienhua Chen, Ien Cheng, Hoon Choi, Drue Coles, Christian Collberg, George Collins, Eric Conrad, Peter Csaszar, Paul Dietz, Martin Dietzfelbinger, Scot Drysdale, Patricia Ealy, Yaakov Eisenberg, Michael Ernst, Michael Formann, Nedim Fresko, Hal Gabow, Marek Galecki, Igal Galperin, Luisa Gargano, John Gately, Rosario Genario, Mihaly Gereb, Ronald Greenberg, Jerry Grossman, Stephen Guattery, Alexander Hartemik, Anthony Hill, Thomas Hofmeister, Mathew Hostetter, Yih-Chun Hu, Dick Johnsonbaugh, Marcin Jurdzinki, Nabil Kahale, Fumiaki Kamiya, Anand Kanagala, Mark Kantrowitz, Scott Karlin, Dean Kelley, Sanjay Khanna, Haluk Konuk, Dina Kravets, Jon Kroger, Bradley Kuszmaul, Tim Lambert, Hang Lau, Thomas Lengauer, George Madrid, Bruce Maggs, Victor Miller, Joseph Muskat, Tung Nguyen, Michael Orlov, James Park, Seongbin Park, Ioannis Paschalidis, Boaz Patt-Shamir, Leonid Peshkin, Patricio Poblete, Ira Pohl, Stephen Ponzio, Kjell Post, Todd Poynor, Colin Prepscius, Sholom Rosen, Dale Russell, Hershel Safer, Karen Seidel, Joel Seiferas, Erik Seligman, Stanley Selkow, Jeffrey Shallit, Greg Shannon, Micha Sharir, Sasha Shen, Norman Shulman, Andrew Singer, Daniel Sleator, Bob Sloan, Michael Sofka, Volker Strumpen, Lon Sunshine, Julie Sussman, Asterio Tanaka, Clark Thomborson, Nils Thommesen, Homer Tilton, Martin Tompa, Andrei

Toom, Felzer Torsten, Hirendu Vaishnav, M. Veldhorst, Luca Venuti, Jian Wang, Michael Wellman, Gerry Wiener, Ronald Williams, David Wolfe, Jeff Wong, Richard Woundy, Neal Young, Huaiyuan Yu, Tian Yuxing, Joe Zachary, Steve Zhang, Florian Zschoke, and Uri Zwick.

Many of our colleagues provided thoughtful reviews or filled out a long survey. We thank reviewers Nancy Amato, Jim Aspnes, Kevin Compton, William Evans, Peter Gacs, Michael Goldwasser, Andrzej Proskurowski, Vijaya Ramachandran, and John Reif. We also thank the following people for sending back the survey: James Abello, Josh Benaloh, Bryan Beresford-Smith, Kenneth Blaha, Hans Bodlaender, Richard Borie, Ted Brown, Domenico Cantone, M. Chen, Robert Cimikowski, William Clocksin, Paul Cull, Rick Decker, Matthew Dickerson, Robert Douglas, Margaret Fleck, Michael Goodrich, Susanne Hambrusch, Dean Hendrix, Richard Johnsonbaugh, Kyriakos Kalorkoti, Srinivas Kankanahalli, Hikyoo Koh, Steven Lindell, Errol Lloyd, Andy Lopez, Dian Rae Lopez, George Lucker, David Maier, Charles Martel, Xiannong Meng, David Mount, Alberto Policriti, Andrzej Proskurowski, Kirk Pruhs, Yves Robert, Guna Seetharaman, Stanley Selkow, Robert Sloan, Charles Steele, Gerard Tel, Murali Varanasi, Bernd Walter, and Alden Wright. We wish we could have carried out all your suggestions. The only problem is that if we had, the second edition would have been about 3000 pages long!

The second edition was produced in LaTeX 2ε. Michael Downes converted the LaTeX macros from "classic" LaTeX to LaTeX 2ε, and he converted the text files to use these new macros. David Jones also provided LaTeX 2ε support. Figures for the second edition were produced by the authors using MacDraw Pro. As in the first edition, the index was compiled using Windex, a C program written by the authors, and the bibliography was prepared using BibTeX. Ayorkor Mills-Tettey and Rob Leathern helped convert the figures to MacDraw Pro, and Ayorkor also checked our bibliography.

As it was in the first edition, working with The MIT Press and McGraw-Hill has been a delight. Our editors, Bob Prior of The MIT Press and Betsy Jones of McGraw-Hill, put up with our antics and kept us going with carrots and sticks.

Finally, we thank our wives-Nicole Cormen, Gail Rivest, and Rebecca Ivry-our children-Ricky, William, and Debby Leiserson; Alex and Christopher Rivest; and Molly, Noah, and Benjamin Stein-and our parents-Renee and Perry Cormen, Jean and Mark Leiserson, Shirley and Lloyd Rivest, and Irene and Ira Stein-for their love and support during the writing of this book. The patience and encouragement of our families made this project possible. We affectionately dedicate this book to them.

THOMAS H. CORMEN
Hanover, New Hampshire

CHARLES E. LEISERSON
Cambridge, Massachusetts

RONALD L. RIVEST
Cambridge, Massachusetts

CLIFFORD STEIN
Hanover, New Hampshire

Prove that the flows in a network form a ***convex set***. That is, show that if f_1 and f_2 are flows, then so is $\alpha f_1 + (1 - \alpha) f_2$ for all α in the range $0 \le \alpha \le 1$.

[redacted]

Exercises 26.1-8

[redacted]

State the maximum-flow problem as a linear-programming problem.

[redacted]

Exercises 26.1-9

[redacted]

Professor Adam has two children who, unfortunately, dislike each other. The problem is so severe that not only do they refuse to walk to school together, but in fact each one refuses to walk on any block that the other child has stepped on that day. The children have no problem with their paths crossing at a corner. Fortunately both the professor's house and the school are on corners, but beyond that he is not sure if it is going to be possible to send both of his children to the same school. The professor has a map of his town. Show how to formulate the problem of determining if both his children can go to the same school as a maximum-flow problem.

26.2 The Ford-Fulkerson method

This section presents the Ford-Fulkerson method for solving the maximum-flow problem. We call it a "method" rather than an "algorithm" because it encompasses several implementations with differing running times. The Ford-Fulkerson method depends on three important ideas that transcend the method and are relevant to many flow algorithms and problems: residual networks, augmenting paths, and cuts. These ideas are essential to the important max-flow min-cut theorem (Theorem 26.7), which characterizes the value of a maximum flow in terms of cuts of the flow network. We end this section by presenting one specific implementation of the Ford-Fulkerson method and analyzing its running time.

The Ford-Fulkerson method is iterative. We start with $f(u, v) = 0$ for all $u, v \in V$, giving an initial flow of value 0. At each iteration, we increase the flow value by finding an "augmenting path," which we can think of simply as a path from the source s to the sink t along which we can send more flow, and then augmenting the flow along this path. We repeat this process until no augmenting path can be found. The max-flow min-cut theorem will show that upon termination, this process yields a maximum flow.

```
FORD-FULKERSON-METHOD(G, s, t)
1  initialize flow f to 0
2  while there exists an augmenting path p
3      do augment flow f along p
4  return f
```

Residual networks

Intuitively, given a flow network and a flow, the residual network consists of edges that can admit more flow. More formally, suppose that we have a flow network $G = (V, E)$ with source s and sink t. Let f be a flow in G, and consider a pair of vertices $u, v \in V$. The amount of *additional* flow we can push from u to v before exceeding the capacity $c(u, v)$ is the ***residual capacity*** of (u, v), given by

$$c_f(u, v) = c(u, v) - f(u, v) . \quad (26.5)$$

For example, if $c(u, v) = 16$ and $f(u, v) = 11$, then we can increase $f(u, v)$ by $c_f(u, v) = 5$ units before we exceed the capacity constraint on edge (u, v). When the flow $f(u, v)$ is negative, the residual capacity $c_f(u, v)$ is greater than the capacity $c(u, v)$. For example, if $c(u, v) = 16$ and $f(u, v) = -4$, then the residual capacity $c_f(u, v)$ is 20. We can interpret this situation as follows. There is a flow of 4 units from v to u, which we can cancel by pushing a flow of 4 units from u to v. We can then push another 16 units from u to v before violating the capacity constraint on edge (u, v). We have thus pushed an additional 20 units of flow, starting with a flow $f(u, v) = -4$, before reaching the capacity constraint.

Given a flow network $G = (V, E)$ and a flow f, the ***residual network*** of G induced by f is $G_f = (V, E_f)$, where

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\} .$$

That is, as promised above, each edge of the residual network, or ***residual edge***, can admit a flow that is greater than 0. Figure 26.3(a) repeats the flow network G and flow f of Figure 26.1(b), and Figure 26.3(b) shows the corresponding residual network G_f.

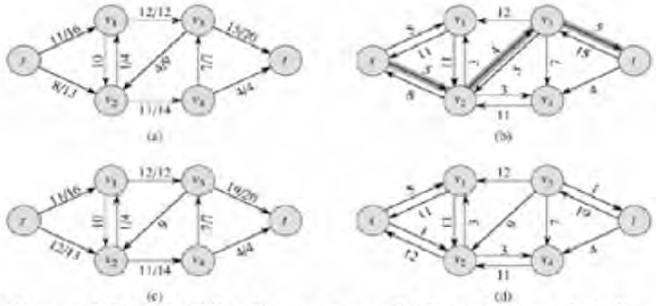

Figure 26.3: *(a)* The flow network G and flow f of Figure 26.1(b). *(b)* The residual network G_f with augmenting path p shaded; its residual capacity is $c_f(p) = c(v_2, v_3) = 4$. *(c)* The flow in G that results from augmenting along path p by its residual capacity 4. *(d)* The residual network induced by the flow in (c).

The edges in E_f are either edges in E or their reversals. If $f(u, v) < c(u, v)$ for an edge $(u, v) \in E$, then $c_f(u, v) = c(u, v) - f(u, v) > 0$ and $(u, v) \in E_f$. If $f(u, v) > 0$ for an edge $(u, v) \in E$, then $f(v, u) < 0$. In this case, $c_f(v, u) = c(v, u) - f(v, u) > 0$, and so $(v, u) \in E_f$. If neither (u, v) nor (v, u) appears in the original network, then $c(u, v) = c(v, u) = 0$, $f(u, v) = f(v, u) = 0$ (by Exercise 26.1-1), and $c_f(u, v) = c_f(v, u) = 0$. We conclude that an edge (u, v) can appear in a residual network only if at least one of (u, v) and (v, u) appears in the original network, and thus

$$|E_f| \le 2 |E| .$$

Observe that the residual network G_f is itself a flow network with capacities given by c_f. The following lemma shows how a flow in a residual network relates to a flow in the original flow network.

Lemma 26.2

Let $G = (V, E)$ be a flow network with source s and sink t, and let f be a flow in G. Let G_f be the residual network of G induced by f, and let f' be a flow in G_f. Then the flow sum $f + f'$ defined by equation (26.4) is a flow in G with value $|f + f'| = |f| + |f'|$.

Proof We must verify that skew symmetry, the capacity constraints, and flow conservation are obeyed. For skew symmetry, note that for all $u, v \in V$, we have

$$\begin{aligned}(f + f')(u, v) &= f(u, v) + f'(u, v) \\ &= -f(v, u) - f'(v, u) \\ &= -(f(v, u) + f'(v, u)) \\ &= -(f + f')(v, u).\end{aligned}$$

For the capacity constraints, note that $f'(u, v) \le c_f(u, v)$ for all $u, v \in V$. By equation (26.5), therefore,

$$\begin{aligned}(f + f')(u, v) &= f(u, v) + f'(u, v) \\ &\le f(u, v) + (c(u, v) - f(u, v)) \\ &= c(u, v).\end{aligned}$$

For flow conservation, note that for all $u \in V - \{s, t\}$, we have

$$\begin{aligned}\sum_{v \in V} (f + f')(u, v) &= \sum_{v \in V} (f(u, v) + f'(u, v)) \\ &= \sum_{v \in V} f(u, v) + \sum_{v \in V} f'(u, v) \\ &= 0 + 0 \\ &= 0 .\end{aligned}$$

Finally, we have

$$\begin{aligned}|f + f'| &= \sum_{v \in V} (f + f')(s, v) \\ &= \sum_{v \in V} (f(s, v) + f'(s, v)) \\ &= \sum_{v \in V} f(s, v) + \sum_{v \in V} f'(s, v) \\ &= |f| + |f'| .\end{aligned}$$

Augmenting paths

Given a flow network $G = (V, E)$ and a flow f, an ***augmenting path*** p is a simple path from s to t in the residual network G_f. By the definition of the residual network, each edge (u, v) on an augmenting path admits some additional positive flow from u to v without violating the capacity constraint on the edge.

The shaded path in Figure 26.3(b) is an augmenting path. Treating the residual network G_f in the figure as a flow network, we can increase the flow through each edge of this path by up to 4 units without violating a capacity constraint, since the smallest residual capacity on this path is $c_f(v_2, v_3) = 4$. We call the maximum amount by which we can increase the flow on each edge in an augmenting path p the ***residual capacity*** of p, given by

$$c_f(p) = \min \{c_f(u, v) : (u, v) \text{ is on } p\}.$$

The following lemma, whose proof is left as Exercise 26.2-7, makes the above argument more precise.

Lemma 26.3

Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Define a function $f_p : V \times V \to \mathbf{R}$ by

$$f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \text{ is on } p, \\ -c_f(p) & \text{if } (v, u) \text{ is on } p, \\ 0 & \text{otherwise.} \end{cases} \tag{26.6}$$

Then, f_p is a flow in G_f with value $|f_p| = c_f(p) > 0$.

The following corollary shows that if we add f_p to f, we get another flow in G whose value is closer to the maximum. Figure 26.3(c) shows the result of adding f_p in Figure 26.3(b) to f from Figure 26.3(a).

Corollary 26.4

Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Let f_p be defined as in equation (26.6). Define a function $f' : V \times V \to \mathbf{R}$ by $f' = f + f_p$. Then f' is a flow in G with value $|f'| = |f| + |f_p| > |f|$.

Proof Immediate from Lemmas 26.2 and 26.3.

Cuts of flow networks

The Ford-Fulkerson method repeatedly augments the flow along augmenting paths until a maximum flow has been found. The max-flow min-cut theorem, which we shall prove shortly, tells us that a flow is maximum if and only if its residual network contains no augmenting path. To prove this theorem, though, we must first explore the notion of a cut of a flow network.

A ***cut*** (S, T) of flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \notin T$. (This definition is similar to the definition of "cut" that we used for minimum spanning trees in Chapter 23, except that here we are cutting a directed graph rather than an undirected graph, and we insist that $s \in S$ and $t \notin T$.) If f is a flow, then the ***net flow*** across the cut (S, T) is defined to be $f(S, T)$. The ***capacity*** of the cut (S, T) is $c(S, T)$. A ***minimum cut*** of a network is a cut whose capacity is minimum over all cuts of the network.

Figure 26.4 shows the cut $(\{s, v_1, v_2\}, \{v_3, v_4, t\})$ in the flow network of Figure 26.1(b). The net flow across this cut is

$$\begin{aligned} f(v_1, v_3) + f(v_2, v_3) + f(v_2, v_4) &= 12 + (-4) + 11 \\ &= 19, \end{aligned}$$

and its capacity is

$$\begin{aligned} c(v_1, v_3) + c(v_2, v_4) &= 12 + 14 \\ &= 26. \end{aligned}$$

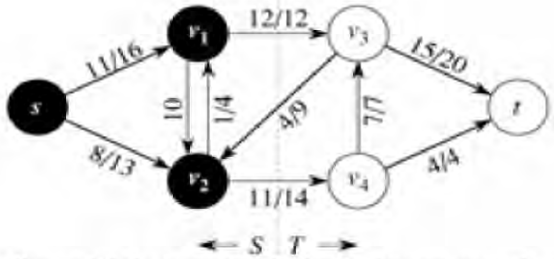

Figure 26.4: A cut (S, T) in the flow network of Figure 26.1(b), where $S = \{s, v_1, v_2\}$ and $T = \{v_3, v_4, t\}$. The vertices in S are black, and the vertices in T are white. The net flow across (S, T) is $f(S, T) = 19$, and the capacity is $c(S, T) = 26$.

Observe that the net flow across a cut can include negative flows between vertices, but that the capacity of a cut is composed entirely of nonnegative values. In other words, the net flow across a cut (S, T) consists of positive flows in both directions; positive flow from S to T is added while positive flow from T to S is subtracted. On the other hand, the capacity of a cut (S, T) is computed only from edges going from S to T. Edges going from T to S are not included in the computation of $c(S, T)$.

The following lemma shows that the net flow across any cut is the same, and it equals the value of the flow.

Lemma 26.5

Let f be a flow in a flow network G with source s and sink t, and let (S, T) be a cut of G. Then the net flow across (S, T) is $f(S, T) = |f|$.

Proof Noting that $f(S - s, V) = 0$ by flow conservation, we have

$$\begin{aligned}
f(S, T) &= f(S, V) - f(S, S) && \text{(by Lemma 26.1, part (3))} \\
&= f(S, V) && \text{(by Lemma 26.1, part (1))} \\
&= f(s, V) + f(S - s, V) && \text{(by Lemma 26.1, part (3))} \\
&= f(s, V) && \text{(since } f(S - s, V) = 0\text{)} \\
&= |f|. &&
\end{aligned}$$

An immediate corollary to Lemma 26.5 is the result we proved earlier-equation (26.3)-that the value of a flow is the total flow into the sink.

Another corollary to Lemma 26.5 shows how cut capacities can be used to bound the value of a flow.

Corollary 26.6

The value of any flow f in a flow network G is bounded from above by the capacity of any cut of G.

Proof Let (S, T) be any cut of G and let f be any flow. By Lemma 26.5 and the capacity constraints,

$$\begin{aligned}
|f| &= f(S, T) \\
&= \sum_{u \in S} \sum_{v \in T} f(u, v) \\
&\le \sum_{u \in S} \sum_{v \in T} c(u, v) \\
&= c(S, T) .
\end{aligned}$$

An immediate consequence of Corollary 26.6 is that the maximum flow in a network is bounded above by the capacity of a minimum cut of the network. The important max-flow min-cut theorem, which we now state and prove, says that the value of a maximum flow is in fact equal to the capacity of a minimum cut.

Theorem 26.7: (Max-flow min-cut theorem)

If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

1. f is a maximum flow in G.
2. The residual network G_f contains no augmenting paths.
3. $|f| = c(S, T)$ for some cut (S, T) of G.

Proof (1) □ (2): Suppose for the sake of contradiction that f is a maximum flow in G but that G_f has an augmenting path p. Then, by Corollary 26.4, the flow sum $f + f_p$, where f_p is given by equation (26.6), is a flow in G with value strictly greater than $|f|$, contradicting the assumption that f is a maximum flow.

(2) □ (3): Suppose that G_f has no augmenting path, that is, that G_f contains no path from s to t. Define

$S = \{v \in V$: there exists a path from s to v in $G_f\}$

and $T = V - S$. The partition (S, T) is a cut: we have $s \in S$ trivially and $t \notin S$ be cause there is no path from s to t in G_f. For each pair of vertices u and v such that $u \in S$ and $v \in T$, we have $f(u, v) = c(u, v)$, since otherwise $(u, v) \in E_f$, which would place v in set S. By Lemma 26.5, therefore, $|f| = f(S, T) = c(S, T)$.

(3) □ (1): By Corollary 26.6, $|f| \leq c(S, T)$ for all cuts (S, T). The condition $|f| = c(S, T)$ thus implies that f is a maximum flow.

[redacted]

The basic Ford-Fulkerson algorithm

In each iteration of the Ford-Fulkerson method, we find *some* augmenting path p and increase the flow f on each edge of p by the residual capacity $c_f(p)$. The following implementation of the method computes the maximum flow in a graph $G = (V, E)$ by updating the flow $f[u, v]$ between each pair u, v of vertices that are connected by an edge.[1] If u and v are not connected by an edge in either direction, we assume implicitly that $f[u, v] = 0$. The capacities $c(u, v)$ are assumed to be given along with the graph, and $c(u, v) = 0$ if $(u, v) \notin E$. The residual capacity $c_f(u, v)$ is computed in accordance with the formula (26.5). The expression $c_f(p)$ in the code is actually just a temporary variable that stores the residual capacity of the path p.

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ E[G]
2       do f[u, v] ← 0
3          f[v, u] ← 0
4  while there exists a path p from s to t in the residual network G_f
5       do c_f(p) ← min {c_f(u, v) : (u, v) is in p}
6          for each edge (u, v) in p
7               do f[u, v] ← f[u, v] + c_f(p)
8                  f[v, u] ← -f[u, v]
```

The FORD-FULKERSON algorithm simply expands on the FORD-FULKERSON-METHOD pseudocode given earlier. Figure 26.5 shows the result of each iteration in a sample run. Lines 1-3 initialize the flow f to 0. The **while** loop of lines 4-8 repeatedly finds an augmenting path p in G_f and augments flow f along p by the residual capacity $c_f(p)$. When no augmenting paths exist, the flow f is a maximum flow.

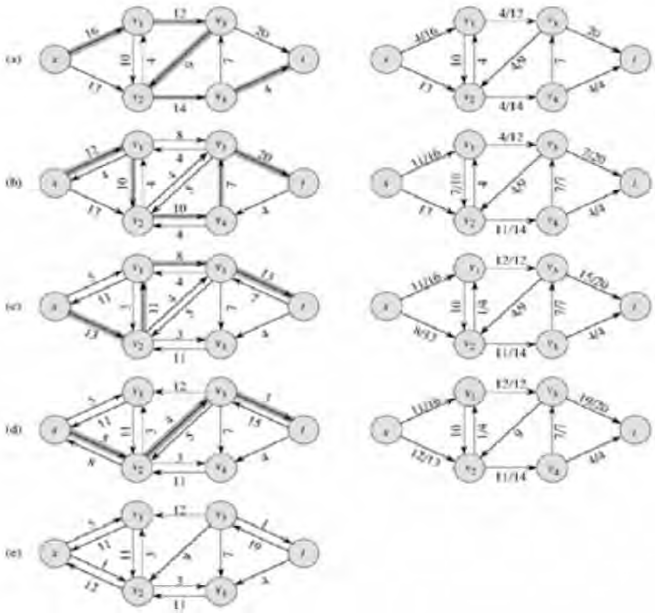

Figure 26.5: The execution of the basic Ford-Fulkerson algorithm. *(a)-(d)* Successive iterations of the *while* loop. The left side of each part shows the residual network G_f from line 4 with a shaded augmenting path p. The right side of each part shows the new flow f that results from adding f_p to f. The residual network in (a) is the input network G. *(e)* The residual network at the last *while* loop test. It has no augmenting paths, and the flow f shown in (d) is therefore a maximum flow.

Analysis of Ford-Fulkerson

The running time of FORD-FULKERSON depends on how the augmenting path p in line 4 is determined. If it is chosen poorly, the algorithm might not even terminate: the value of the flow will increase with successive augmentations, but it need not even converge to the maximum flow value.[2] If the augmenting path is chosen by using a breadth-first search (which we saw in Section 22.2), however, the algorithm runs in polynomial time. Before proving this result, however, we obtain a simple bound for the case in which the augmenting path is chosen arbitrarily and all capacities are integers.

Most often in practice, the maximum-flow problem arises with integral capacities. If the capacities are rational numbers, an appropriate scaling transformation can be used to make them all integral. Under this assumption, a straightforward implementation of FORD-FULKERSON runs in time $O(E\,|f^*|)$, where f^* is the maximum flow found by the algorithm. The analysis is as follows. Lines 1-3 take time $\Theta(E)$. The **while** loop of lines 4-8 is executed at most $|f^*|$ times, since the flow value increases by at least one unit in each iteration.

The work done within the **while** loop can be made efficient if we efficiently manage the data structure used to implement the network $G = (V, E)$. Let us assume that we keep a data structure corresponding to a directed graph $G' = (V, E')$, where $E' = \{(u, v) : (u, v) \in E \text{ or } (v, u) \in E\}$. Edges in the network G are also edges in G', and it is therefore a simple matter to maintain capacities and flows in this data structure. Given a flow f on G, the edges in the residual network G_f consist of all edges (u, v) of G' such that $c(u, v) - f[u, v] \neq 0$. The time to find a path in a residual network is therefore $O(V + E') = O(E)$ if we use either depth-first search or breadth-first search. Each iteration of the **while** loop thus takes $O(E)$ time, making the total running time of FORD-FULKERSON $O(E\,|f^*|)$.

When the capacities are integral and the optimal flow value $|f^*|$ is small, the running time of the Ford-Fulkerson algorithm is good. Figure 26.6(a) shows an example of what can happen on a simple flow network for which $|f^*|$ is large. A maximum flow in this network has value 2,000,000: 1,000,000 units of flow traverse the path $s \rightarrow u \rightarrow t$, and another 1,000,000 units traverse the path $s \rightarrow v \rightarrow t$. If the first augmenting path found by FORD-FULKERSON is $s \rightarrow u \rightarrow v \rightarrow t$, shown in Figure 26.6(a), the flow has value 1 after the first iteration. The resulting residual network is shown in Figure 26.6(b). If the second iteration finds the augmenting path $s \rightarrow v \rightarrow u \rightarrow t$, as shown in Figure 26.6(b), the flow then has value 2. Figure 26.6(c) shows the resulting residual network. We can continue, choosing the augmenting path $s \rightarrow u \rightarrow v \rightarrow t$ in the odd-numbered iterations and the augmenting path $s \rightarrow v \rightarrow u \rightarrow t$ in the even-numbered iterations. We would perform a total of 2,000,000 augmentations, increasing the flow value by only 1 unit in each.

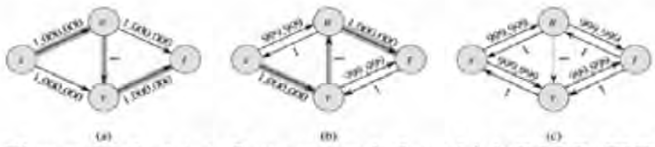

Figure 26.6: *(a)* A flow network for which FORD-FULKERSON can take $\Theta(E\,|f^*|)$ time, where f^* is a maximum flow, shown here with $|f^*| = 2{,}000{,}000$. An augmenting path with residual capacity 1 is shown. *(b)* The resulting residual network. Another augmenting path with residual capacity 1 is shown. *(c)* The resulting residual network.

The Edmonds-Karp algorithm

The bound on FORD-FULKERSON can be improved if we implement the computation of the augmenting path p in line 4 with a breadth-first search, that is, if the augmenting path is a *shortest* path from s to t in the residual network, where each edge has unit distance (weight). We call the Ford-Fulkerson method so implemented the ***Edmonds-Karp algorithm***. We now prove that the Edmonds-Karp algorithm runs in $O(V E^2)$ time.

The analysis depends on the distances to vertices in the residual network G_f. The following lemma uses the notation $\delta_f(u, v)$ for the shortest-path distance from u to v in G_f, where each edge has unit distance.

Lemma 26.8

[redacted]

If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and sink t, then for all vertices $v \in V - \{s, t\}$, the shortest-path distance $\delta_f(s, v)$ in the residual network G_f increases monotonically with each flow augmentation.

Proof We will suppose that for some vertex $v \in V - \{s, t\}$, there is a flow augmentation that causes the shortest-path distance from s to v to decrease, and then we will derive a contradiction. Let f be the flow just before the first augmentation that decreases some shortest-path distance, and let f' be the flow just afterward. Let v be the vertex with the minimum $\delta_{f'}(s, v)$ whose distance was decreased by the augmentation, so that $\delta_{f'}(s, v) < \delta_f(s, v)$. Let $p = s \rightsquigarrow u \rightarrow v$ be a shortest path from s to v in $G_{f'}$, so that $(u, v) \in E_{f'}$ and

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1 . \tag{26.7}$$

Because of how we chose v, we know that the distance label of vertex u did not decrease, i.e.,

$$\delta_{f'}(s, u) \geq \delta_f(s, u) . \tag{26.8}$$

We claim that $(u, v) \notin E_f$. Why? If we had $(u, v) \in E_f$, then we would also have

$$\begin{aligned} \delta_f(s, v) &\leq \delta_f(s, u) + 1 && \text{(by Lemma 24.10, the triangle inequality)} \\ &\leq \delta_{f'}(s, u) + 1 && \text{(by inequality (26.8))} \\ &= \delta_{f'}(s, v) && \text{(by equation (26.7)),} \end{aligned}$$

which contradicts our assumption that $\delta_{f'}(s, v) < \delta_f(s, v)$.

How can we have $(u, v) \notin E_f$ and $(u, v) \in E_{f'}$? The augmentation must have increased the flow from v to u. The Edmonds-Karp algorithm always augments flow along shortest paths, and therefore the shortest path from s to u in G_f has (v, u) as its last edge. Therefore,

$$\begin{aligned} \delta_f(s, v) &= \delta_f(s, u) - 1 \\ &\leq \delta_{f'}(s, u) - 2 && \text{(by inequality (26.8))} \\ &= \delta_{f'}(s, v) - 2 && \text{(by equation (26.7)),} \end{aligned}$$

which contradicts our assumption that $\delta_{f'}(s, v) < \delta_f(s, v)$. We conclude that our assumption that such a vertex v exists is incorrect.

The next theorem bounds the number of iterations of the Edmonds-Karp algorithm.

Theorem 26.9

If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and sink t, then the total number of flow augmentations performed by the algorithm is $O(V E)$.

Proof We say that an edge (u, v) in a residual network G_f is ***critical*** on an augmenting path p if the residual capacity of p is the residual capacity of (u, v), that is, if $c_f(p) = c_f(u, v)$. After we have augmented flow along an augmenting path, any critical edge on the path disappears from the residual network. Moreover, at least one edge on any augmenting path must be critical. We will show that each of the $|E|$ edges can become critical at most $|V|/2 - 1$ times.

Let u and v be vertices in V that are connected by an edge in E. Since augmenting paths are shortest paths, when (u, v) is critical for the first time, we have

$$\delta_f(s, v) = \delta_f(s, u) + 1.$$

Once the flow is augmented, the edge (u, v) disappears from the residual network. It cannot reappear later on another augmenting path until after the flow from u to v is decreased, which occurs only if (v, u) appears on an augmenting path. If f' is the flow in G when this event occurs, then we have

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1.$$

Since $\delta_f(s, v) \leq \delta_{f'}(s, v)$ by Lemma 26.8, we have

$$\begin{aligned}\delta_{f'}(s, u) &= \delta_{f'}(s, v) + 1 \\ &\geq \delta_f(s, v) + 1 \\ &= \delta_f(s, u) + 2.\end{aligned}$$

Consequently, from the time (u, v) becomes critical to the time when it next becomes critical, the distance of u from the source increases by at least 2. The distance of u from the source is initially at least 0. The intermediate vertices on a shortest path from s to u cannot contain s, u, or t (since (u, v) on the critical path implies that $u \neq t$). Therefore, until u becomes unreachable from the source, if ever, its distance is at most $|V| - 2$. Thus, (u, v) can become critical at most $(|V|-2)/2 = |V|/2-1$ times. Since there are $O(E)$ pairs of vertices that can have an edge between them in a residual graph, the total number of critical edges during the entire execution of the Edmonds-Karp algorithm is $O(VE)$. Each augmenting path has at least one critical edge, and hence the theorem follows.

[redacted]

Since each iteration of FORD-FULKERSON can be implemented in $O(E)$ time when the augmenting path is found by breadth-first search, the total running time of the Edmonds-Karp algorithm is $O(VE^2)$. We shall see that push-relabel algorithms can yield even better bounds. The algorithm of Section 26.4 gives a method for achieving an $O(V^2E)$ running time, which forms the basis for the $O(V^3)$-time algorithm of Section 26.5.

Exercises 26.2-1

[redacted]

In Figure 26.1(b), what is the flow across the cut $(\{s, v_2, v_4\}, \{v_1, v_3, t\})$? What is the capacity of this cut?

[redacted]

Exercises 26.2-2

[redacted]

Show the execution of the Edmonds-Karp algorithm on the flow network of Figure 26.1(a).

[redacted]

Exercises 26.2-3

[redacted]

In the example of Figure 26.5, what is the minimum cut corresponding to the maximum flow shown? Of the augmenting paths appearing in the example, which two cancel flow?

[redacted]

Exercises 26.2-4

[redacted]

Prove that for any pair of vertices u and v and any capacity and flow functions c and f, we have $c_f(u, v) + c_f(v, u) = c(u, v) + c(v, u)$.

[redacted]

Exercises 26.2-5

[redacted]

Recall that the construction in Section 26.1 that converts a multisource, multisink flow network into a single-source, single-sink network adds edges with infinite capacity. Prove that any flow in the resulting network has a finite value if the edges of the original multisource, multisink network have finite capacity.

[redacted]

Exercises 26.2-6

[redacted]

Suppose that each source s_i in a multisource, multisink problem produces exactly p_i units of flow, so that $f(s_i, V) = p_i$. Suppose also that each sink t_j consumes exactly q_j units, so that $f(V, t_j) = q_j$, where $\Sigma_i p_i = \Sigma_j q_j$. Show how to convert the problem of finding a flow f that obeys these additional constraints into the problem of finding a maximum flow in a single-source, single-sink flow network.

[redacted]

Exercises 26.2-7

[redacted]

Prove Lemma 26.3.

[redacted]

Exercises 26.2-8

[redacted]

Show that a maximum flow in a network $G = (V, E)$ can always be found by a sequence of at most $|E|$ augmenting paths. (*Hint:* Determine the paths *after* finding the maximum flow.)

[redacted]

Exercises 26.2-9

[redacted]

The ***edge connectivity*** of an undirected graph is the minimum number k of edges that must be removed to disconnect the graph. For example, the edge connectivity of a tree is 1, and the edge connectivity of a cyclic chain of vertices is 2. Show how the edge connectivity of an undirected graph $G = (V, E)$ can be determined by running a maximum-flow algorithm on at most $|V|$ flow networks, each having $O(V)$ vertices and $O(E)$ edges.

[redacted]

Exercises 26.2-10

[redacted]

Suppose that a flow network $G = (V, E)$ has symmetric edges, that is, $(u, v) \in E$ if and only if $(v, u) \in E$. Show that the Edmonds-Karp algorithm terminates after at most $|V|\,|E|/4$ iterations. (*Hint:* For any edge (u, v), consider how both $\delta(s, u)$ and $\delta(v, t)$ change between times at which (u, v) is critical.)

[redacted]

[1]We use square brackets when we treat an identifier-such as *f*-as a mutable field, and we use parentheses when we treat it as a function.

[2]The Ford-Fulkerson method might fail to terminate only if edge capacities are irrational numbers. In practice, however, irrational numbers cannot be stored on finite-precision computers.

26.3 Maximum bipartite matching

Some combinatorial problems can easily be cast as maximum-flow problems. The multiple-source, multiple-sink maximum-flow problem from Section 26.1 gave us one example. There are other combinatorial problems that seem on the surface to have little to do with flow networks, but can in fact be reduced to maximum-flow problems. This section presents one such problem: finding a maximum matching in a bipartite graph (see Section B.4). In order to solve this problem, we shall take advantage of an integrality property provided by the Ford-Fulkerson method. We shall also see that the Ford-Fulkerson method can be made to solve the maximum-bipartite-matching problem on a graph $G = (V, E)$ in $O(V E)$ time.

The maximum-bipartite-matching problem

Given an undirected graph $G = (V, E)$, a ***matching*** is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v. We say that a vertex $v \in V$ is ***matched*** by matching M if some edge in M is incident on v; otherwise, v is ***unmatched***. A ***maximum***

Show that the following linear program is unbounded:

maximize $x_1 - x_2$
subject to

$$\begin{aligned} -2x_1 + x_2 &\le -1 \\ -x_1 - 2x_2 &\le -2 \\ x_1, x_2 &\ge 0 \end{aligned}$$

.

[redacted]

Exercises 29.1-8

[redacted]

Suppose that we have a general linear program with n variables and m constraints, and suppose that we convert it into standard from. Give an upper bound on the number of variables and constraints in the resulting linear program.

[redacted]

Exercises 29.1-9

[redacted]

Give an example of a linear program for which the feasible region is not bounded, but the optimal objective value is finite.

29.2 Formulating problems as linear programs

Although we shall focus on the simplex algorithm in this chapter, it is also important to be able to recognize when a problem can be formulated as a linear program. Once a problem is formulated as a polynomial-sized linear program, it can be solved in polynomial time by the ellipsoid or interior-point algorithms. Several linear-programming software packages can solve problems efficiently, so that once the problem has been expressed as a linear program, it can be solved in practice by such a package.

We shall look at several concrete examples of linear-programming problems. We start with two problems that we have already studied: the single-source shortest-paths problem (see Chapter 24) and the maximum-flow problem (see Chapter 26). We then describe the minimum-cost-flow problem. There is a polynomial-time algorithm that is not based on linear programming for the minimum-cost-flow problem, but we shall not examine it. Finally, we describe the multicommodity-flow problem, for which the only known polynomial-time algorithm is based on linear programming.

Shortest paths

The single-source shortest-paths problem, described in Chapter 24, can be formulated as a linear program. In this section, we shall focus on the formulation of the single-pair shortest-path problem, leaving the extension to the more general single-source shortest-paths problem as Exercise 29.2-3.

In the single-pair shortest-path problem, we are given a weighted, directed graph $G = (V, E)$, with weight function $w : E \to \mathbf{R}$ mapping edges to real-valued weights, a source vertex s, and a destination vertex t. We wish to compute the value $d[t]$, which is the weight of a shortest path from s to t. To express this problem as a linear program, we need to determine a set of variables and constraints that define when we have a shortest path from s to t. Fortunately, the Bellman-Ford algorithm does exactly this. When the Bellman-Ford algorithm terminates, it has computed, for each vertex v, a value $d[v]$ such that for each edge $(u, v) \in E$, we have $d[v] \leq d[u] + w(u, v)$. The source vertex initially receives a value $d[s] = 0$, which is never changed. Thus we obtain the following linear program to compute the shortest-path weight from s to t:

(29.44) **minimize** $d[t]$

subject to

(29.45) $d[v] \leq d[u] + w(u, v)$ for each edge $(u, v) \in E$,
(29.46) $d[s] = 0$.

In this linear program, there are $|V|$ variables $d[v]$, one for each vertex $v \in V$. There are $|E| + 1$ constraints, one for each edge plus the additional constraint that the source vertex always has the value 0.

Maximum flow

The maximum-flow problem can also be expressed as a linear program. Recall that we are given a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$, and two distinguished vertices, a sink s and a source t. As defined in Section 26.1, a flow is a real-valued function $f : V \times V \to \mathbf{R}$ that satisfies three properties: capacity constraints, skew symmetry, and flow conservation. A maximum flow is a flow that satisfies these constraints and maximizes the flow value, which is the total flow coming out of the source. A flow, therefore, satisfies linear constraints, and the value of a flow is a linear function. Recalling also that we assume that $c(u, v) = 0$ if $(u, v) \notin E$, we can express the maximum-flow problem as a linear program:

(29.47) **maximize** $\sum_{v \in V} f(s, v)$

subject to

(29.48) $f(u, v) \leq c(u, v)$ for each $u, v \in V$,
(29.49) $f(u, v) = -f(v, u)$ for each $u, v \in V$,
(29.50) $\sum_{v \in V} f(u, v) = 0$ for each $u \in V - \{s, t\}$.

This linear program has $|V|^2$ variables, corresponding to the flow between each pair of vertices, and it has $2|V|^2 + |V| - 2$ constraints.

It is usually more efficient to solve a smaller-sized linear program. The linear program in (29.47)–(29.50) has, for ease of notation, a flow and capacity of 0 for each pair of vertices u, v with $(u, v) \notin E$. It would be more efficient to rewrite the linear program so that it has $O(V + E)$ constraints. Exercise 29.2-5 asks you to do so.

Minimum-cost flow

In this section, we have used linear programming to solve problems for which we already knew efficient algorithms. In fact, an efficient algorithm designed specifically for a problem, such as Dijkstra's algorithm for the single-source shortest-paths problem, or the push-relabel method for maximum flow, will often be more efficient than linear programming, both in theory and in practice.

The real power of linear programming comes from the ability to solve new problems. Recall the problem faced by the politician in the beginning of this chapter. The problem of obtaining a sufficient number of votes, while not spending too much money, is not solved by any of the algorithms that we have studied in this book, yet it is solved by linear programming. Books abound with such real-world problems that linear programming can solve. Linear programming is also particularly useful for solving variants of problems for which we may not already know of an efficient algorithm.

Consider, for example, the following generalization of the maximum-flow problem. Suppose that each edge (u, v) has, in addition to a capacity $c(u, v)$, a real-valued cost $a(u, v)$. If we send $f(u, v)$ units of flow over edge (u, v), we incur a cost of $a(u, v) f(u, v)$. We are also given a flow target d. We wish to send d units of flow from s to t in such a way that the total cost incurred by the flow, $\Sigma_{(u, v) \in E} a(u, v) f(u v)$, is minimized. This problem is known as the ***minimum-cost-flow problem***.

Figure 29.3(a) shows an example of the minimum-cost-flow problem. We wish to send 4 units of flow from s to t, while incurring the minimum total cost. Any particular legal flow, that is, a function f satisfying constraints (29.48)–(29.50), incurs a total cost of $\Sigma_{(u, v) \in E} a(u, v) f(u, v)$. We wish to find the particular 4-unit flow that minimizes this cost. An optimal solution is given in Figure 29.3(b), and it has total cost $\Sigma_{(u, v) \in E} a(u, v) f(u, v) =$ $(2 \cdot 2)+(5 \cdot 2)+(3 \cdot 1)+(7 \cdot 1)+(1 \cdot 3) = 27$.

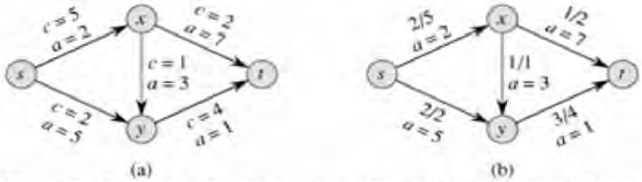

Figure 29.3: *(a)* An example of a minimum-cost-flow problem. We denote the capacities by c and the costs by a. Vertex s is the source and vertex t is the sink, and we wish to send 4 units of flow from s to t. *(b)* A solution to the minimum-cost flow problem in which 4 units of flow are sent from s to t. For each edge, the flow and capacity are written as flow/capacity.

There are polynomial-time algorithms specifically designed for the minimum-cost-flow problem, but they are beyond the scope of this book. We can, however, express the minimum-cost-flow problem as a linear program. The linear program looks similar to the one for the maximum-flow problem with the additional constraint that the value of the flow be exactly d units, and with the new objective function of minimizing the cost:

(29.51) minimize $\sum_{(u, v) \in E} a(u, v) f(u, v)$

subject to

$$
\begin{aligned}
&(29.52)\quad f(u,v) \le c(u,v) \quad \text{for each } u, v \in V ,\\
&(29.53)\quad f(u,v) = -f(v,u) \quad \text{for each } u, v \in V ,\\
&(29.54)\quad \sum_{v \in V} f(u,v) = 0 \quad \text{for each } u \in V - \{s,t\} ,\\
&(29.55)\quad \sum_{v \in V} f(s,v) = d .
\end{aligned}
$$

Multicommodity flow

As a final example, we consider another flow problem. Suppose that the Lucky Puck company from Section 26.1 decides to diversify its product line and ship not only hockey pucks, but also hockey sticks and hockey helmets. Each piece of equipment is manufactured in its own factory, has its own warehouse, and must be shipped, each day, from factory to warehouse. The sticks are manufactured in Vancouver and must be shipped to Saskatoon, and the helmets are manufactured in Edmonton and must be shipped to Regina. The capacity of the shipping network does not change, however, and the different items, or ***commodities***, must share the same network.

This example is an instance of a ***multicommodity-flow problem***. In this problem, we are again given a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \ge 0$. As in the maximum-flow problem, we implicitly assume that $c(u, v) = 0$ for $(u, v) \notin E$. In addition, we are given k different commodities, $K_1, K_2, \ldots, K_k$, where commodity i is specified by the triple $K_i = (s_i, t_i, d_i)$. Here, s_i is the source of commodity i, t_i is the sink of commodity i, and d_i is the demand, which is the desired flow value for commodity i from s_i to t_i. We define a flow for commodity i, denoted by f_i, (so that $f_i(u, v)$ is the flow of commodity i from vertex u to vertex v) to be a real-valued function that satisfies the flow-conservation, skew-symmetry, and capacity constraints. We now define $f(u, v)$, the ***aggregate flow***, to be sum of the various commodity flows, so that $f(u, v) = \sum_{i=1}^{k} f_i(u, v)$. The aggregate flow on edge (u, v) must be no more than the capacity of edge (u, v). This constraint subsumes the capacity constraints for the individual commodities. The way this problem is described, there is nothing to minimize; we need only determine whether it is possible to find such a flow. Thus, we write a linear program with a "null" objective function:

minimize 0

subject to

$$
\begin{aligned}
&\sum_{i=1}^{k} f_i(u,v) \le c(u,v) && \text{for each } u, v \in V ,\\
&f_i(u,v) = -f_i(v,u) && \text{for each } i = 1, 2, \ldots, k \text{ and}\\
&&& \text{for each } u, v \in V ,\\
&\sum_{v \in V} f_i(u,v) = 0 && \text{for each } i = 1, 2, \ldots, k \text{ and}\\
&&& \text{for each } u \in V - \{s_i, t_i\} ,\\
&\sum_{v \in V} f_i(s,v) = d_i && \text{for each } i = 1, 2, \ldots, k .
\end{aligned}
$$

The only known polynomial-time algorithm for this problem is to express it as a linear program and then solve with a polynomial-time linear-programming algorithm.

Exercises 29.2-1